Vol. 41 No. 3 Transactions of Information Processing Society of Japan Mar. 2000

Regular Paper

Aspect-centered Design of Object-oriented Frameworks

Shin Nakajima†

This paper reports experience in developing object-oriented frameworks for an implementation of the OMG trading object service. To bridge the gap between the real world complex problem and existing object-oriented methods, the trading function is first analyzed to identify a set of aspects, each introducing a subproblem. The subproblem is *heterogeneous* in the sense that it is associated with a particular specification technique to reach a solution. All the resultant solutions form a whole design artifact that is the input to the object-oriented design phase. The phase produces a *homogeneous* object solution by using existing object-oriented design methods such as collaboration-based design and design patterns. The paper also identifies two further research topics, (1) aspect discovery, and (2) checking integrity of all the aspect solutions.

1. Introduction

Object-oriented framework is a promising solution technology for improving reusability of software, where a framework is a reusable design of a program or a part of a program expressed as a set of classes [6),9)]. Having recognized the importance of object-oriented framework, many methodologists propose design methods focusing on framework development. The methods include collaboration-based design [4),5),13)], role-based design [16)] and design patterns [7),9),15)]. However, desiging well-organized frameworks is still an art.

Jackson points out two important issues on method and problem in general [8)]; (1) methods cannot be panaceas (medicines that cure all diseases), and (2) very few problems can be decomposed into *homogeneous* structures. Because real world system is complex, the problem is decomposed into a set of subproblems. The subproblem is *heterogeneous* in the sense that it needs a different problem frame (a kind of structural pattern to solve the subproblem). In addition, methods should be related to a particular class of problem and thus give a sharply focused help in reaching a solution.

In developing object-oriented frameworks for a real world complex system, a new design method is necessary. The method bridges the gap between the complex problem and existing object-oriented methods; the problem is one such that is decomposed into a set of heterogeneous subproblems, and the object-oriented methods can handle only homogeneous world of objects. The new design method focuses on decomposing the whole problem into a set of simple subproblems. Each subproblem needs not to be object-oriented, but is associated with specification technique best fitted for the intrinsic nature of the subproblem. The design process continues to produce homogeneous object solution by using existing object-oriented design methods.

This paper reports experience in developing object-oriented frameworks for an implementation of the OMG trading object service [1)]. Before using the known techniques of developing object-oriented frameworks, the trading function is first analyzed in order to identify a set of distinct aspects☆. Then, the subproblem associated with each aspect is refined and elaborated by using the specification technique fitted for the subproblem. Although each technique is not new, the main contribution of the paper is (1) to propose that the idea of *aspect* is a way to bridge the gap, and (2) to show that the proposed method is effective in developing frameworks to implement the OMG trading server (a non-trivial distributed service).

2. Aspect-Centered Design Method

Existing design methods for developing object-oriented frameworks, such as the collaboration-based design [4),5),13)] and design pattern [7),9),15)], are effective in general. However, the methods have several drawbacks. (1) The design methods have their basis on the

† NEC C&C Media Research Laboratories

☆ The terminology, *aspect*, is borrowed from Kiczales, et al.[10)] because viewing a software system consisting of many aspects is the common idea.

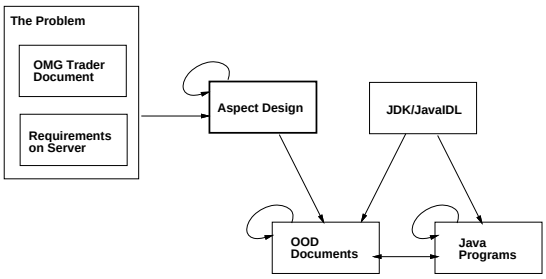

Fig. 1 Design process.

object-orientation, and explicitly assume that *object* is sole constituent of the system. (2) The methods provide only general guidelines of decomposing a whole problem into constituent objects, and mention no concrete hint for the decomposition. (3) The design pattern is a catalog of useful design idioms, but most of them are at a programming level and are thus not suitable for use at an early stage of the development.

Real world software such as the OMG trader server is a complex system. As will be discussed in Section 3, the target problem has various aspects that are not amenable to *homogeneous* object-oriented modeling. Analyzing the target problem and decomposing it into a set of subproblems is the most important task.

Figure 1 summarizes the design process adapted in the present approach. The first step of the process (the aspect design phase) is identifying a set of distinct aspects in the problem to obtain a semi-formal description. The phase starts with analyzing both the OMG document and the system requirement. Using a specification technique best fitted for the characteristics of each aspect reaches aspect solutions. The solutions form a whole design artifact that is the input to the next phase. The phase (object-oriented design) makes use of existing methods such as collaboration-based design or design patterns. In the course of preparing the design document, some part of the framework is implemented incrementally in Java [3].

The following two characteristics of the aspect design seems well-known common practice; (1) decomposing a large complex problem into a set of manageable subpoblems to solve individually, and (2) seeing a target system from various viewpoints. For example, a top-down functional design approach deals with decomposition into procedures or processes. Its decomposition is homogeneous and hierarchical. OMT provides three *models* (object, dynamic, and functional), and promotes a method to describe the system behavior by using the three different models [17]. The model, however, represents a different viewpoint of a same entitiy, *object*.

On the other hand, the important characteristics of the aspect-centered design method is *heterogeneity*. *Aspect* is related to a subproblem that is further refined and elaborated to reach solution description individually. The subproblem needs not to be object-oriented, but is *heterogeneous* in the sense that each subproblem is associated with specification technique best fitted for its intrinsic nature.

A simple example on Composite pattern [7] may illustrate the above discussion. Composite pattern is a design pattern to represent tree structures. When a target system has an aspect of language processing such as a kind of query language, Composite pattern together with Visitor pattern is quite useful in implementing language processing subsystem☆. However, BNF is a better notation to discuss the grammatical aspects of the problem such as abstract syntax of the language. BNF is more concise than class diagrams representing the abstract syntax tree. Usually BNF is used in comparing various designs, and then the patterns are employed to instantiate appropriate classes. Because other part of the given problem may use notation different from BNF, the problem can be said to be decomposed into a set of *heterogeneous* subproblems. And, the resultant classes are *homogeneous* representations.

Since identifying aspects in the target problem is not a well-established methodology, the present paper relies on a case study. The presented result may not be generally applicable, but provides a useful insight on the application of the method because the case study talks about the problem in a concrete manner. The rest of the paper presents experience of a case study in applying the aspect design method to the development of object-oriented frameworks to implement the OMG trading server and discusses the pros and cons of the proposed approach.

☆ Section 4.5 illustrates how to use Composite and Visitor patterns in implementing an aspect of the trading server.

3. Trading Object Service

3.1 Trading Functions

Figure 2 shows a trader and participants in a trading scenario [1),18)]. The service server (`Exporter`) exports a service offer. The service client (`Importer`) imports the service offer and then invokes the service. The `Trader` mediates between the two by using the exported service offers stored in its repository that is ready for import requests. The `Service Type` provides information necessary to define service offers, and holds the interface type of the object being advertised and a list of property definitions. Because importing is the most complex and interesting function, this paper focuses on the importing action.

The OMG trading service comprises five major functional interfaces; `Lookup`, `Register`, `Admin`, `Link`, and `Proxy`. The OMG standard also specifies six different conformance classes of trading object service implementation. A *simple trader* supports the `Lookup` and `Register` interfaces, and a *linked trader* adds the `Admin` and `Link` interfaces to the simple trader. The following IDL fragment shows a portion of a `query` operation in the `Lookup` interface[☆].

```
typedef Istring ServiceTypeName;
typedef Istring Constraint;
typedef Istring Preference;

void query(
  in ServiceTypeName type,
  in Constraint constr,
  in Preference pref,
  ...
  out OfferSeq offers,
  out OfferIterator offer_itr,
  ...
) raises ( ... )
```

The first parameter `type` specifies the service type name of requested offers. The parameter `constr` denotes a condition that the offers should satisfy and is an expression of a *constraint language*, a means to specify the condition in a concise manner. The expression describes semantically a set of property values of service offers that the client tries to import. The trader searches its repository to find service offers whose property values match the importer's request. The parameter `pref` is preference information specifying that the matched offers are sorted according to the preference rule. The sorted offers are returned to the importer in the `out` parameters (`offers` and `offer_itr`). Additionally, the original specification defines a set of *scoping policies* to define upper bounds (cardinalities) of offers to be searched. The values of the cardinalities are determined by a combination of importer's policies and trader's policies.

[☆] Parameters not relevant here are omitted for brevity.

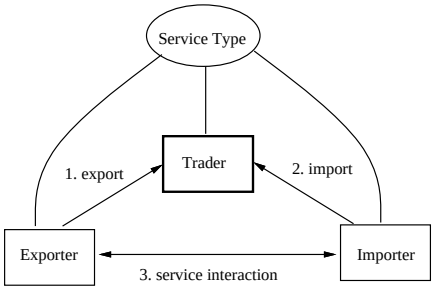

Fig. 2 Trading scenario.

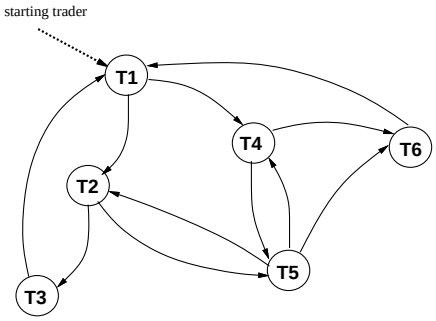

Fig. 3 Federated trader group.

In order to achieve scalability, the OMG trading object service defines the specification for interworking or federation of traders. A federated trader group can partition a large number of service offers into a set of smaller ones of manageable size. One trader is responsible for each partition and works with the other traders when necessary. **Figure 3** is an example of a federated trader group. The traders, T1 to T6, are linked as indicated by the curved arrows. When a `query` is issued, for example, on T1 as a starting trader and federated search is requested, other traders (T2 to T6) also initiate local search respectively. All the matched offers are collected and returned to the client importer.

The federation process uses a set of policies for controlling the graph traversal. A simple one is the `request_id` that cuts unnecessary

visits to the same trader more than once, and the `hop_count` restricts the depth of traders to be visited. A set of policies called the *FollowOption* controls the traversal semantically. A link marked with `if_no_local`, for example, is followed only if no matched offer is found locally in a trader at the source of the link.

Although the OMG standard describes the query algorithm and the role of each scoping policy by using illustrative figures, it is not easy to grasp the whole picture. The reason is that the descriptions are informal and scattered over several pages of the document [1].

3.2 Architectural Considerations

The OMG trading service consists of five major functional interfaces and some of their specific combinations correspond to different conformance classes. In order to develop in future a series of traders that belong to a different conformance class, subsystems each implementing a particular functional interface are desirable to be separated to form a well-organized architecture.

In addition to satisfying what the OMG document specifies, developing a server to implement the trading function requires designing control architecture. The server should be responsible for handling multiple client requests in order to improve availability of the trading service. **Figure 4** presents an abstract view of the trader, which focuses on subsystems concerning both local and federated query processing. The trader should have a resource management mechanism because some repository objects storing `ServiceType` or `ServiceOffer` become shared resources. Such control aspects is desirable to be decoupled from the rest of the trader functionalities.

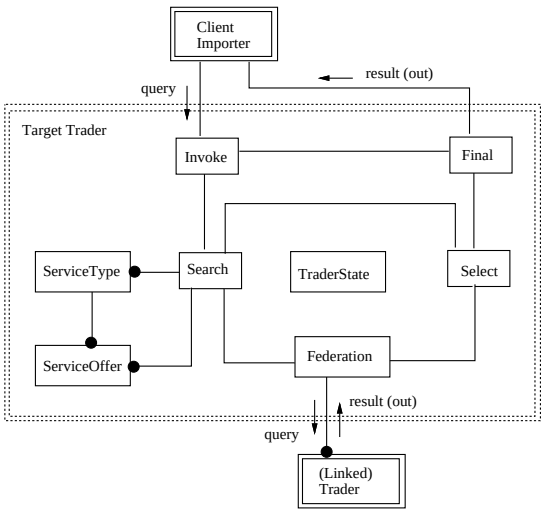

Fig. 4 Subsystem structure.

4. Design and Implementation

4.1 Overview of Identified Aspects

Table 1 summarizes the identified aspects together with the accompanying specification techniques. Of the entries in Table 1, the common concepts such as `ServiceType` and `PropertyDefinition` are easily translated into class definitions since these are modeled as abstract datatypes. Both the architecture and functional objects are refined and elaborated by using collaboration-based design methods [4),13)]; collaboration diagrams or message sequence charts are used to analyze their interaction patterns so that the responsibility of each participant object is well defined. However, modeling the control aspects of the resouce management requires detailed knowledge of the middleware solution used and the execution mechanism that the implementation language/library provides, Java [3] and JavaIDL [11] in this case.

The succeeding subsections will discuss the other important aspects including language, policy and algorithm (see Table 1).

4.2 Query Algorithm and Policy

The *policy* of the trading service is just a parameter that modifies behavior of both local and federated query algorithm. It is hard to understand the meaning of policies without referring to basic algorithm. In addition, in order to grasp the algorithm at a glance, a concise notation is needed. The notation adapted is the one borrowed from a functional programming language StandardML [12] augmented with some symbols to describe and handle set-like collections of data. Another important decision here is a choice of a stream-style functional programming for the query algorithm. This viewpoint is in accordance with the informal presentation in the OMG document [1].

The following subsections will explain in detail the descriptions that actually appear in the appendix of the paper. Each function is refered

Table 1 Aspects and specification techniques.

Aspect	Specification Technique
language	denotational semantics
policy	functional programming
algorithm	stream-style programming
common concept	abstract datatype
architecture	collaboration-based design
functional object	collaboration-based design

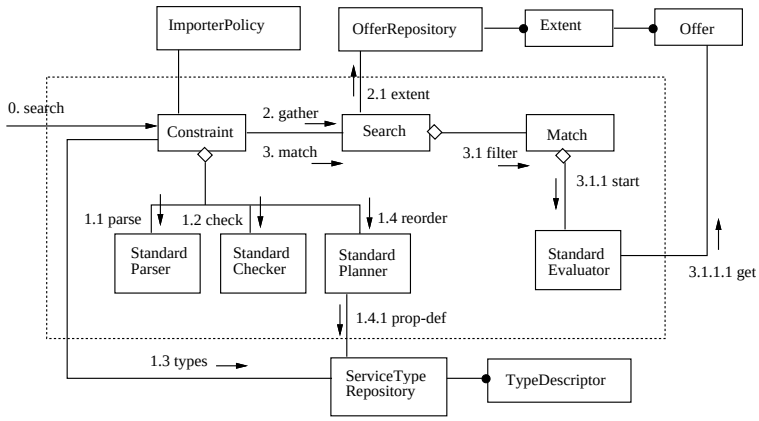

Fig. 5 Constraint language processing framework.

to by an index number such as (♯1).

4.3 Local Query

This subsection deals with the query algorithm that is executed locally in a trader.

The top-level function `IDLquery(T,I)` (♯1), which is invoked as an IDL request takes the form below. All the function definitions are supposed to come in the lexical context (as `fun` $\cdots$) of the `IDLquery(T,I)`. The functions can use `T` and `I` freely as global constants, where `T` refers to the trader state or trader's policy and `I` denotes the importer's request and policy.

```
fun IDLquery(T,I) =
  fun query() = if valid-trader()
    then if valid_id() then
      (select ∘ federation ∘ search)(T.offers)
      else φ
    else IDLquery(remote_trader(T),I)
  fun ···
  in
    query()
  end
```

First, it checks whether the request is on the trader itself. Then, it invokes the body of the `query` function, which is described as a stream-style processing consisting of `search`, `federation`, and `select`.

The function `search` (♯2) is responsible for collecting candidate offers. The candidate space is then truncated by appropriate policies on cardinality. The `search` uses two such cardinality filters.

The function `gather` (♯3) collects offers that have the specified service type. If the importer policy has a false `I.exact_type_match`, offers of all the subtypes of the specified one should be collected (♯4). In the definition of (♯5), the content of TypeRepository is a directed acyclic graph (`G`) whose node (`n`) is a service type and edge is a service subtype relationship ($\prec$).

The function `match` returns offers that satisfy the importer's requirement expressed as a constraint language expression. Its representation will be discussed in Section 4.5.

The next two functions (♯6 and ♯7) implement filtering on cardinality mentioned above. Both use the `truncate` function to filter out unnecessary offers. The two functions represent how to compute each cardinality in a concise manner. The role of the policy concerning the cardinality is thus clear.

When a federated query is not in use, the function invoked after the `search` is the `select` (♯8). With a help of `order` (♯9), it uses the importer's preference expression to make the collected offers into a sequence, the sequence of which reflects the preference order.

At this point, analyzing all the behavior of local query is completed. Next, an object-oriented framework, based on the description of the algorithm, is elaborated. Translating the description into design of object-oriented framework is not difficult because the stream-style description can be considered as an abstract representation of collaboration. The stream-style description becomes the input specification of the *frozen spots* of the framework and is a good starting point for identifying *hot spots*.

A main design activity is decomposing the whole behavior into a set of participant classes. As shown in **Fig. 5**, the main function of the algorithm is distributed over `Search`, `Match`, and `StandardEvaluator` classes. The stream-style processing sequence is translated into the internal flow of control that the

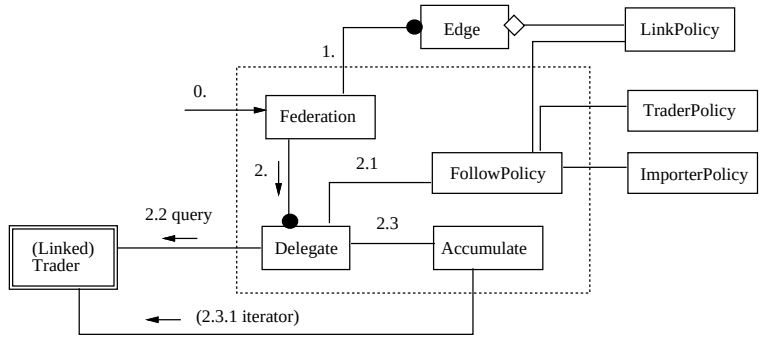

Fig. 6 Federation framework.

framework encapsulates. Some objects such as `ServiceTypeRepository`, `ImporterPolicy` and `OfferRepository` provide necessary data for the processing.

4.4 Federated Query

This subsection deals with the federated query algorithm that involves more than one trader.

The function `federation(R)` (♯11) is responsible for controling a federated query. It first checks whether further IDL query requests are necessary to linked traders by consulting the trader's policy on `hop_count`. The auxiliary function `new_hop_count()` (♯12) demonstrates the role of policies involved in the federation control.

The function `traversal` (♯13) is invoked with a modified importer policy (`J`) and the offers obtained locally (`R`). It controls invocations on the target trader located at the far end of the specified link. The control again requires a scoping policy calculation, which involves the link policies as well as the trader's policies and the importer's. The two functions `new_importer_follow_rule(L,J)` (♯14) and `current_link_follow_rule(L,J)` (♯15) show the definitions of `FollowOption` rule. The function `dispatch` (♯16) concisely gives specifications of the use of the `FollowOption` rule of the specified link; the rule defines three cases, `local_only`, `if_no_local`, and `always`. How to construct the final offers differs in each case.

Figure 6 shows the framework that implements the federation process. Some of the important decisions include encapsulating FollowOption calculation functions in class `FollowPolicy` and separating functionality among `Federation` and `Delegate`. Class `Delegate` corresponds to the body of the function `traversal(J,R)` and thus implements details of the algorithm. Class `Federation` is responsible for controlling the whole federation process and plays a role of Façade [7]. It decouples the federation subsystem from the rest, and thus makes it easy for testing.

```
CExp ::= Pred
Pred ::=  L
        | Exp == Exp
        | exist L
        | not Pred
        | Pred and Pred
        | Pred or Pred
        | ...
```

Fig. 7 Abstract syntax (a part).

4.5 Constraint Language Processing

This subsection deals with the aspect of the constraint language processing. The accompanying specification technique is a denotational style of language definition.

Two functions (`order_on_preference` and `match`) used in the main algorithm in Section 4.3 involve evaluation of a constraint expression and a preference expression. Each function is defined to call an evaluation function (either $\mathcal{CE}$ or $\mathcal{PE}$).

fun match(R) = $\mathcal{CE}$ ⟦ I.constraint ⟧ R

fun order_on_preference(R,X) = $\mathcal{PE}$ ⟦ X ⟧ R

The specification technique follows a standard way of rigorous language definition. First, the abstract syntax of the language is introduced. A portion is shown in **Fig. 7**. Second, a valuation function is defined for each syntax category; $\mathcal{CE}$ is an example function for constraint expressions (`CExp`) and it further calls $\mathcal{LE}$ of the valuation function for predicates (`Pred`). `R` stands for a set of offers and `O` is an offer.

$\mathcal{CE}$: CExp → R → R

$\mathcal{LE}$: Pred → O → Bool

Then, the specifications of the constraint language interpreter or evaluator are best seen by the definitions of the valuation function. The definitions can be formulated systematically by

studying the meaning of each abstract syntax construct.

$\mathcal{CE}[\![\ \mathrm{E}\ ]\!]\ \mathrm{R} = \{\ \mathrm{O} \in \mathrm{R} \mid \mathcal{LE}[\![\ \mathrm{E}\ ]\!]\ \mathrm{O}\ \}$
$\mathcal{LE}[\![\ \mathrm{L}\ ]\!]\ \mathrm{O} = \text{prop-val}(\mathrm{O},\mathrm{L})\downarrow_{Bool}$
$\mathcal{LE}[\![\ \mathrm{E1} == \mathrm{E2}\ ]\!]\ \mathrm{O} =$
$\quad \mathcal{AE}[\![\ \mathrm{E1}\ ]\!]\ \mathrm{O} == \mathcal{AE}[\![\ \mathrm{E2}\ ]\!]\ \mathrm{O}$
...

Designing the framework for constraint language processing from the above language definition is straightforward. The design activity makes use of design patterns [7]: Composite pattern for representing the abstract syntax tree (AST) and Visitor pattern for representing the tree walkers such as a light semantic checker (`StandarcChecker`), a filtering condition reorder planner (`StandardPlanner`), and an interpreter (`StandardEvaluator`).

In implementing the constraint language processor, two offline support tools are used to enhance productivity; JavaCC [2](a public domain Java-based parser generator) and ASTG (an in-house visitor skeleton generator). ASTG accepts annotated BNF descriptions of abstract syntax and generates Java class definitions implementing the AST node objects and also skeleton codes for tree walking. The skeleton code follows a convention of a Visitor pattern. Since the program code fragments that need to be written in body part of the skeleton corresponds to the clauses of the valuation functions, completing the program is not difficult.

5. Discussions

The complexity of the OMG trading object service and the architectural considerations motivated us first to identify a set of distinct aspects of the target problem. Of six identified aspects in Table 1, the meaning of policies and the query functions are described in terms of the stream-style functional programming model, and the constraint language is defined and elaborated by means of standard technique for defining language semantics. The aspect solutions form a clear input specification to the framework design phase where existing methods such as collaboration-base design and design patterns are effective.

The aspect solution description is concise and thus expected to ease future maintenance[☆]. For example, the sixteen functions in the appendix abstractly describes the design of about 30 Java classes, and about thirty lines of the denotational style language description becomes more than 4 K lines of Java codes including AST classes and visitor skeletons that ASTG automatically generated. The current prototype implementation of the trading service server consists of some 380 Java classes, and its code size is about 25 K lines[☆☆].

The aspect solution shows clear relationships between the design descriptions at the different levels. First, the stream-style description does not have a large gap with the OMG document, and, thus, both descriptions are quite traceable. It is easy to perform conformance checking during the design phase. Second, collaboration, which is the most important view of frameworks, is basically a set of global interaction patterns and requires a concise notation for grasping the global flow of control. Algorithm description using the functional programming style is a good candidate for such a representation.

The actual development process consisted of three steps, the aspect design, the object-oriented design, and the coding and testing (Fig. 1). The aspect design step started from formalizing the system requirements and ended with the semi-formal descriptions of aspect solutions. The object-oriented design step employed collaboration-based object-oriented design method and design pattern to produce design documents describing Java classes. It was followed by the usual coding and testing step. One person (this author) was responsible for the aspect design and two engineers for the coding and testing. All the three persons worked together to produce the design documents at the intermediate step. The phase helped the engineers to deepen their understanding of the system design. The engineers first resisted the mostly functional style description of the aspect design. The engineers were not familiar with the collaboration-based object-oriented design method and required "on the job" style training in the course of the system development. The object-oriented design step involved technology transfer, and took periods of time far longer than initially planned. However, the coding and testing was short compared with the program

☆ The size is not constant because we maintain and update the program code periodically. The information here is only meant to illustrate the system size.

☆☆ The current prototype implements a linked trader which consists of `Lookup`, `Register`, `Admin`, and `Link` interfaces. It omits the `Proxy` interface.

size.

Two research areas can be identified relating to the aspect-centered design method; (1) aspect discovery, and (2) checking integrity of all the aspect solutions.

The hard part of the aspect-centered design method is lack of systematic methodology to discover appropriate aspects in a given problem. Since each aspect is accompanied with a specific specification technique, knowing specification techniques as many as possible is helpful in identifying aspect in the problem. Accumulating various specification techniques and experience with their application to system development is one of the future directions.

The idea of aspect-centered design is essentially decomposing a whole problem into a set of manageable subproblems to solve individually. Each aspect has its own notation such as functional-style descriptions or message sequence charts, and is amenable to validate separately. On the other hand, the design of the overall system requires to integrate all the solution descriptions, and this is difficult when each aspect solution uses a different notation. Therefore, currently the integration is done only through manual reviews during the design phase of object-oriented framework (Fig. 1).

Concerning the issue on the notation, two approach would be possible; (a) establishing relationship between different notations, and (b) providing a homogeneous notation. An example of the first approach is recent activities on assigning rigorous semantics to various UML diagrams. UML, however, is based on the object-orientation and also not adequate for compact algorithm descriptions. For the second approach, formal notation such as algebraic specification language would be a candidate. It is because the language is powerful enough to cover aspect solution descriptions, from functional programming descriptions and abstract datatypes to message sequence charts [13),14)]. Further research is necessary to show the effectiveness.

6. Summary

This paper presented experience in developing object-oriented frameworks for an implementation of the OMG trading object service by using Java and JavaIDL. In order to bridge the gap between the real world complex problem and existing object-oriented modeling methods, the trading function was analyzed to identify a set of distinct aspects. Subproblem associating with each aspect was refined and elaborated by using specification technique fitted for the subproblem. Then, the design process continued to produce *homogeneous* object solution by using existing object-oriented design methods such collaboration-based design and design patterens. Last, the discussion included the pros and cons of the proposed approach and identified two future research issues.

References

1) OMG: CORBA services, Trading Object Service Specification (1997).
2) Sun Microsystems: JavaCC Documentation (http://www.suntest.com/JavaCC/).
3) Arnold, K. and Gosling, J.: *The Java™ Programming Language*, Addison-Wesley (1996).
4) Beck, K. and Cunningham, W.: A Laboratory for Teaching Object-Oriented Thinking, *Proc. OOPSLA'89*, pp.1–6 (1989).
5) Carroll, J.M. (Ed.): *Scenario-Based Design*, John Wiley & Sons (1995).
6) Deutsch, L.P.: Design Reuse and Frameworks in the Smalltalk-80 Programming System, Biggerstaff and Perlis (Eds.), *Software Reusability*, Vol.2, pp.55–71, ACM Press (1989).
7) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994).
8) Jackson, M.: *Software Requirements & Specifications*, Addison-Wesley (1995).
9) Johnson, R.: Documenting Frameworks using Patterns, *Proc. OOPSLA'92*, pp.63–76 (1992).
10) Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *Proc. ECOOP'97*, pp.220–242 (1997).
11) Lewis, G., Barber, S. and Siegel, E.: *Programming with Java IDL*, John Wiley & Sons (1998).
12) Milner, R., Tofte, M., Harper, R. and MacQueen, D.: *The Definition of Standard ML* (revised), MIT Press (1997).
13) Nakajima, S. and Futatsugi, K.: An Object-Oriented Modeling Method for Algebraic Specifications in CafeOBJ, *Proc. ICSE'97*, pp.34–44 (1997).
14) Nakajima, S.: Using Algebraic Specification Techniques in Development of Object-Oriented Frameworks, *Proc. FM'99*, pp.1664–1683 (1999).
15) Pree, W.: Meta Patterns – A Means for Capturing the Essentials of Reusable Object-Oriented Design, *Proc. ECOOP'94*, pp.150–

Appendix: Functional Programming Style Descriptions

```
(♯2)  fun search(R)
      = (match_cardinality_filter ∘ match ∘ search_cardinality_filter ∘ gather)(R)
(♯3)  fun gather(R) = { s ∈ R | s.ServiceType ∈ requested_types() }
(♯4)  fun requested_types()
      = if I.exact_type_match then { type(I.ServiceTypeName) }
        else collect_subtypes(T.TypeRepository,type(I.ServiceTypeName))
(♯5)  fun collect_subtypes(G,N) = { n ∈ node(G) | n ≺* N }
(♯6)  fun search_cardinality_filter(R)
      = truncate((if exist(I.search_card) then min(I.search_card, T.max_search_card)
                                          else T.def_search_card), R)
(♯7)  fun match_cardinality_filter(R)
      = truncate((if exist(I.match_card) then min(I.match_card, T.max_match_card)
                                         else T.def_match_card), R)
(♯8)  fun select(R) = (return_cardinality_filter ∘ order)(R)
(♯9)  fun order(R) = order_on_preference(R,I.preference)
(♯10) fun return_cardinality_filter(Q)
      = truncate((if exist(I.return_card) then min(I.return_card, T.max_return_card)
                                          else T.def_return_card), R)
(♯11) fun federation(R)
      = let val new_count = new_hop_count()
        in
            if new_count ≥ 0 then traversal((I with new_count),R) else R
        end
(♯12) fun new_hop_count()
      = (if exist(I.hop_count)
         then min(I.hop_count, T.max_hop_count) else T.def_hop_count) - 1
(♯13) fun traversal(J,R)
      = ⋃∀L∈T.links dispatch_on(current_link_follow_rule(L,J), L,
                                (I with new_importer_follow_rule(L,J)),R)
(♯14) fun new_importer_follow_rule(L,J)
      = if exist(J.link_follow_rule)
        then min(J.link_follow_rule, L.limiting_follow_rule, T.max_follow_policy)
        else min(L.def_pass_on_follow_rule, T.max_follow_policy)
(♯15) fun current_link_follow_rule(L,J)
      = if exist(J.link_follow_rule)
        then min(J.link_follow_rule, L.limiting_follow_rule, T.max_follow_policy)
        else min(L.limiting_follow_rule, T.max_follow_policy, T.def_follow_policy)
(♯16) fun dispatch_on(local_only,L,J,R) = R
        | dispatch_on(if_no_local,L,J,R) = if empty(R) then follow(L,J) else R
        | dispatch_on(always,L,J,R) = follow(L,J) ∪ R
(♯17) fun follow(L,J) = IDLquery(L.trader,J)
```


162 (1994).

16) Riehle, D. and Gross, T.: Role Model Based Framework Design and Integration, *Proc. OOPSLA'98*, pp.117–133 (1998).

17) Rumbaugh, J., Blaha, M., Premeriani, W., Eddy, F. and Lorensen, W.: *Object-Oriented Modeling and Design*, Prentice-Hall (1991).

18) Vogel, A. and Duddy, K.: *Java*™ *Programming with CORBA* (2ed.), John Wiley & Sons (1998).

(Received February 12, 1999)
(Accepted December 2, 1999)

Shin Nakajima is affiliated with C&C Media Research Laboratories, NEC Corporation. He received B.A. and M.Sc. degrees in physics from the University of Tokyo in 1979 and 1981 respectively. During 1988–89 academic year, he was a visiting researcher with the University of Oregon. His research interest includes distributed software engineering, algebraic specifications, and meta-level architecture. He is also a visiting lecturer at Tokyo Metropolitan University since 1992.